Diseño y verificación de un multiplicador secuencial con signo mediante el algoritmo de Booth modificado

Tarea 2

Integración de Sistemas Digitales

2025/2026

Estructura de la práctica

1. **Introducción de la práctica.**
   - Objetivos
   - Estructura de la práctica
   - Funcionamiento de multiplicadores secuenciales
2. **Etapa RTL. Diseño**
   - Implementación del Hardware con descripción literal del ASM
     - *Sin particionado controlpath – data path*
3. **Etapa RTL. Descripción del sistema y verificación funcional**
   - Análisis (compilación) del diseño realizado
   - Verificación funcional del diseño realizado
4. **Etapa Lógica. Compilación del sistema y simulación lógica**
   - Síntesis (compilación) del sistema multiplicador.
     - Obtención Recursos utilizados: logic-cells y flip-flops
   - Análisis temporal estático del diseño realizado:
     - Obtención fmax
5. **Verificación lógica BÁSICA del diseño realizado**
6. **Verificación exhaustiva:**
   - RCSG, covergroups, program, interface, class, aserciones
7. **Segmentación**
8. **Implementación hardware**

ENTREGA:
17 nov 2025 10:00

Guión de la práctica y bibliografía

- **Tarea 2** en PoliformaT (pendiente de actualización)

- Para la parte hardware:
  - Información adjuntada en la tarea

- Para el diseño del banco de pruebas debe tenerse en cuenta
  - Bloque de teoría
  - Conceptos de Verificación
    - **Guia paso a paso y con ejemplos en PoliformaT**

PARTE HARDWARE

entendiendo el multiplicador

El algoritmo the Booth

El Algoritmo the Booth

A SIGNED BINARY MULTIPLICATION TECHNIQUE

By ANDREW D. BOOTH

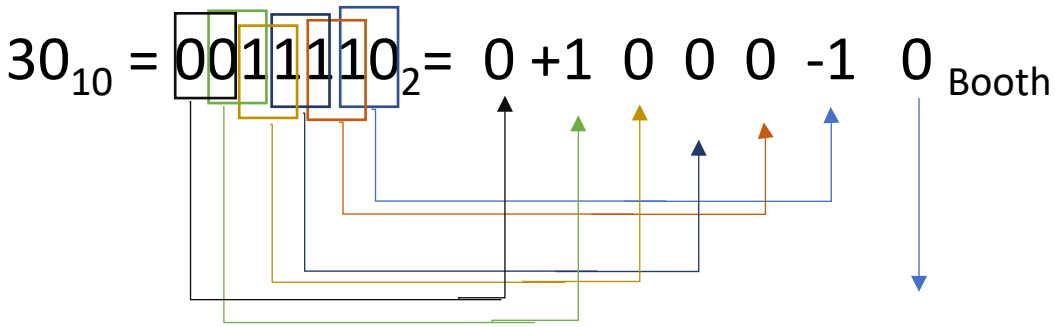
(Birkbeck College Electronic Computer Project, 21 Torrington Square, London, W.C.1)

[Received 1 August 1950]

- El algoritmo de Booth es una alternativa eficiente al clásico algoritmo de multiplicación ADD+SHIFT.
- El algoritmo de Booth consiste en realizar sumas de potencias positivas o negativas de la base en función de parejas de bits (q_i q_{i-1}).

q_i	q_{i-1}	Digito de Booth
0	0	0
0	1	+1
1	0	-1
1	1	0

- El primer paso es recodificar el multiplicador (la base) según la tabla



Debe suponerse un bit implícito 0 a la derecha del LSB

El Algoritmo the Booth

- Funcionamiento:
  - Se multiplica por el numero recodificado de manera que el multiplicando se suma o se resta (sumar el negativo, Ca2) al resultado parcial desplazado a la izquierda.
  - El MSB del resultado parcial se extiende
  - Ejemplo

0	1	0	1	1	0	1	(45_{10})
0	+1	0	0	0	-1	0	(30_{Booth})

Ejemplo algoritmo de Booth

N=4

$M=2_{10}=0010_2$

$Q=-7_{10}=1001_2$

Ciclo	Acción	S - HI - LO - X
0	Carga de valores (LO<=Q ; HI<=0000; S,X<=0	0 0000 1001 0
1	$q_i\ q_{i-1}$ = 10 : HI <- HI –M	1 1110 1001 0
	Desplazamiento	1 1111 0100 1
2	$q_i\ q_{i-1}$ = 01: HI <- HI + M	0 0001 0100 1
	Desplazamiento	0 0000 1010 0
3	$q_i\ q_{i-1}$ = 00: HI <- HI	0 0000 1010 0
	Desplazamiento	0 0000 0101 0
4	$q_i\ q_{i-1}$ = 10: HI <- HI –M	1 1110 0101 0
	Desplazamiento	1 1111 0010 1

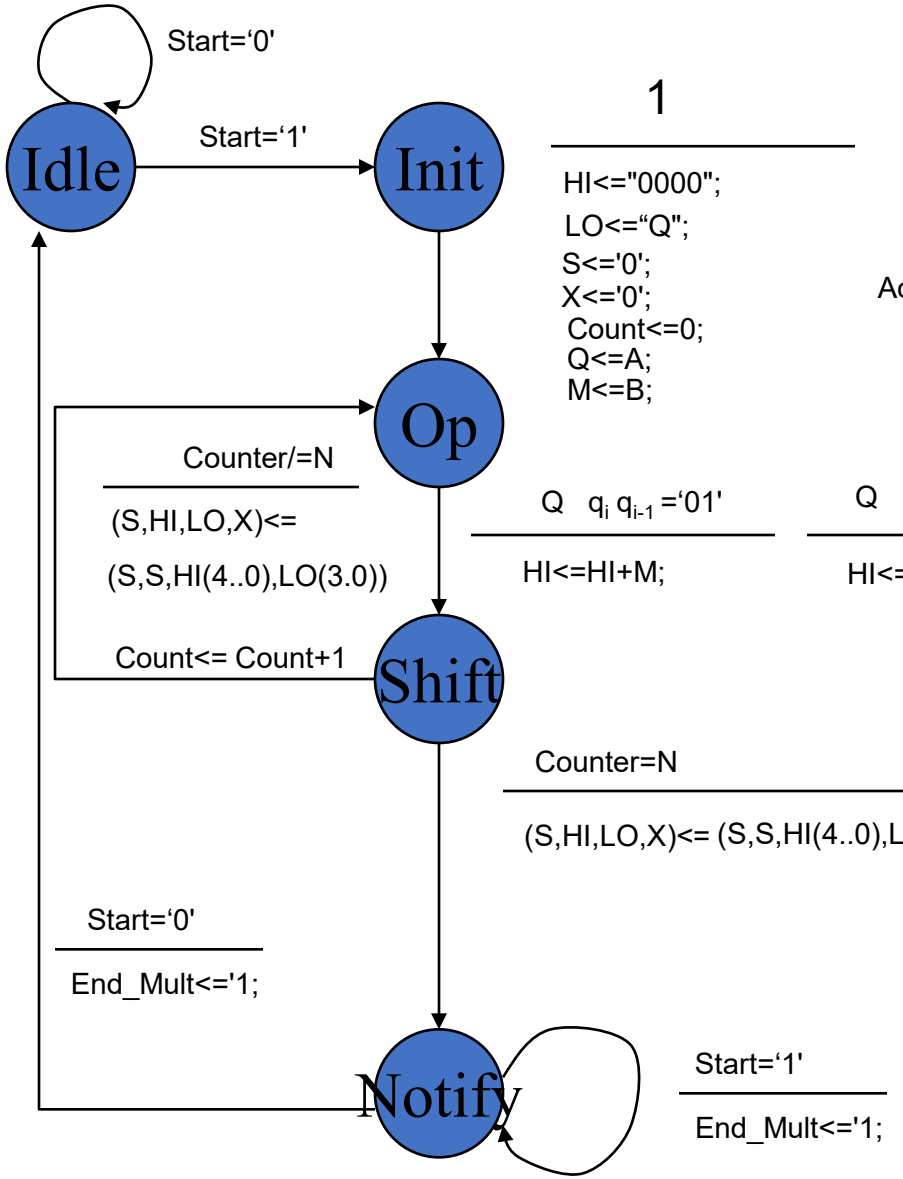
con extensión

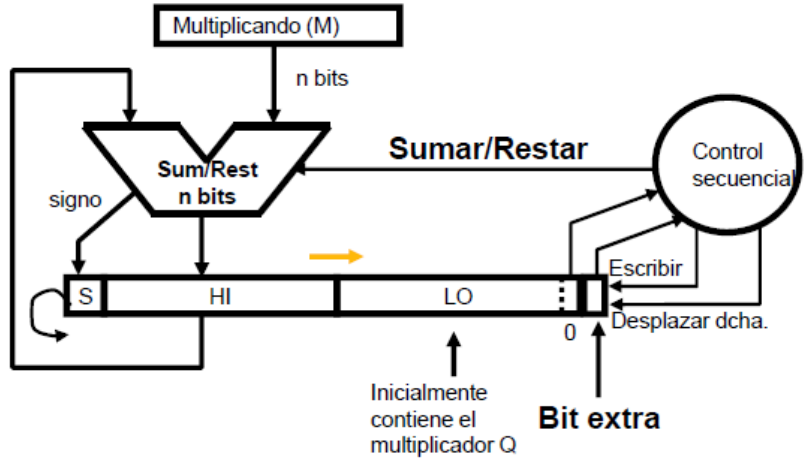
$$\begin{array}{cccccc} & 0 & 0 & 0 & 0 \\ - & 0 & 0 & 1 & 0 \\ \hline 1 & 1 & 1 & 1 & 0 \end{array}$$

$$\begin{array}{cccccc} & 1 & 1 & 1 & 1 \\ + & 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 \end{array}$$

-14_{10}

FSM Algoritmo the Booth





ASM Algoritmo the Booth

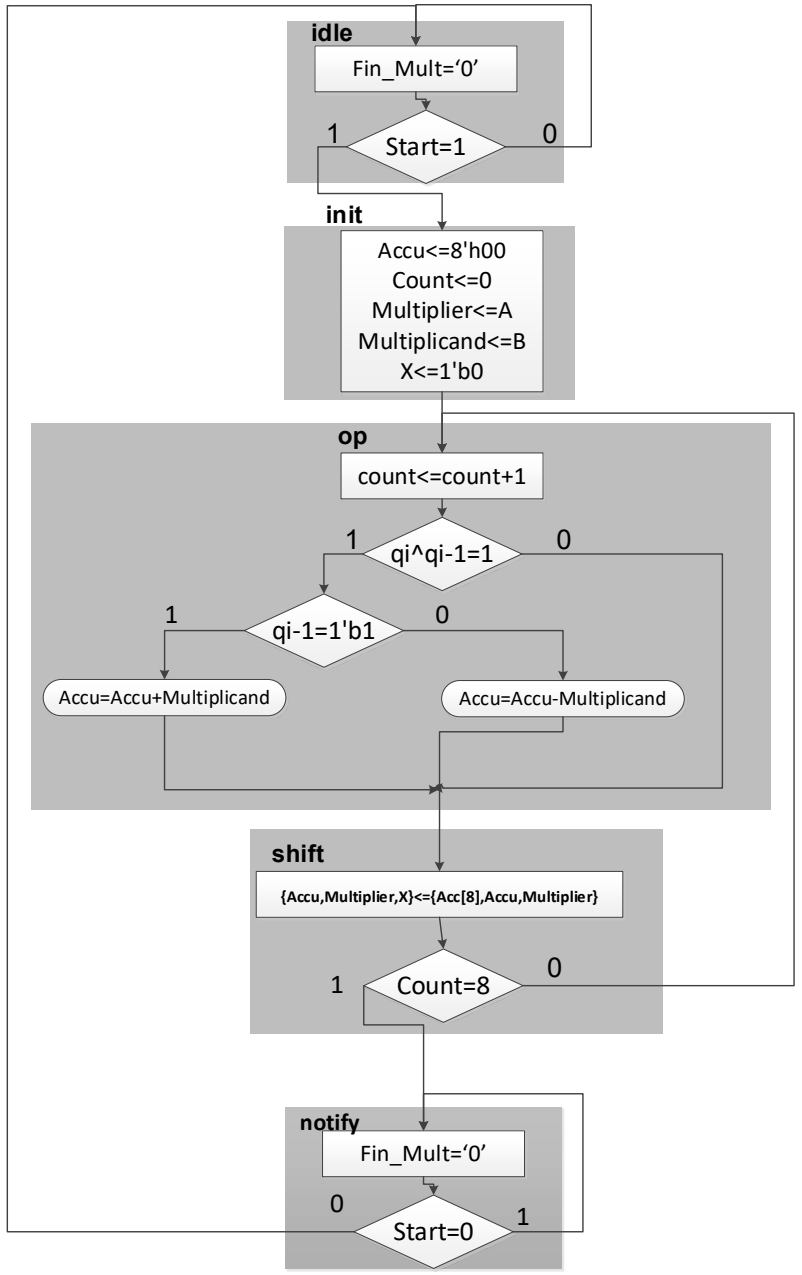
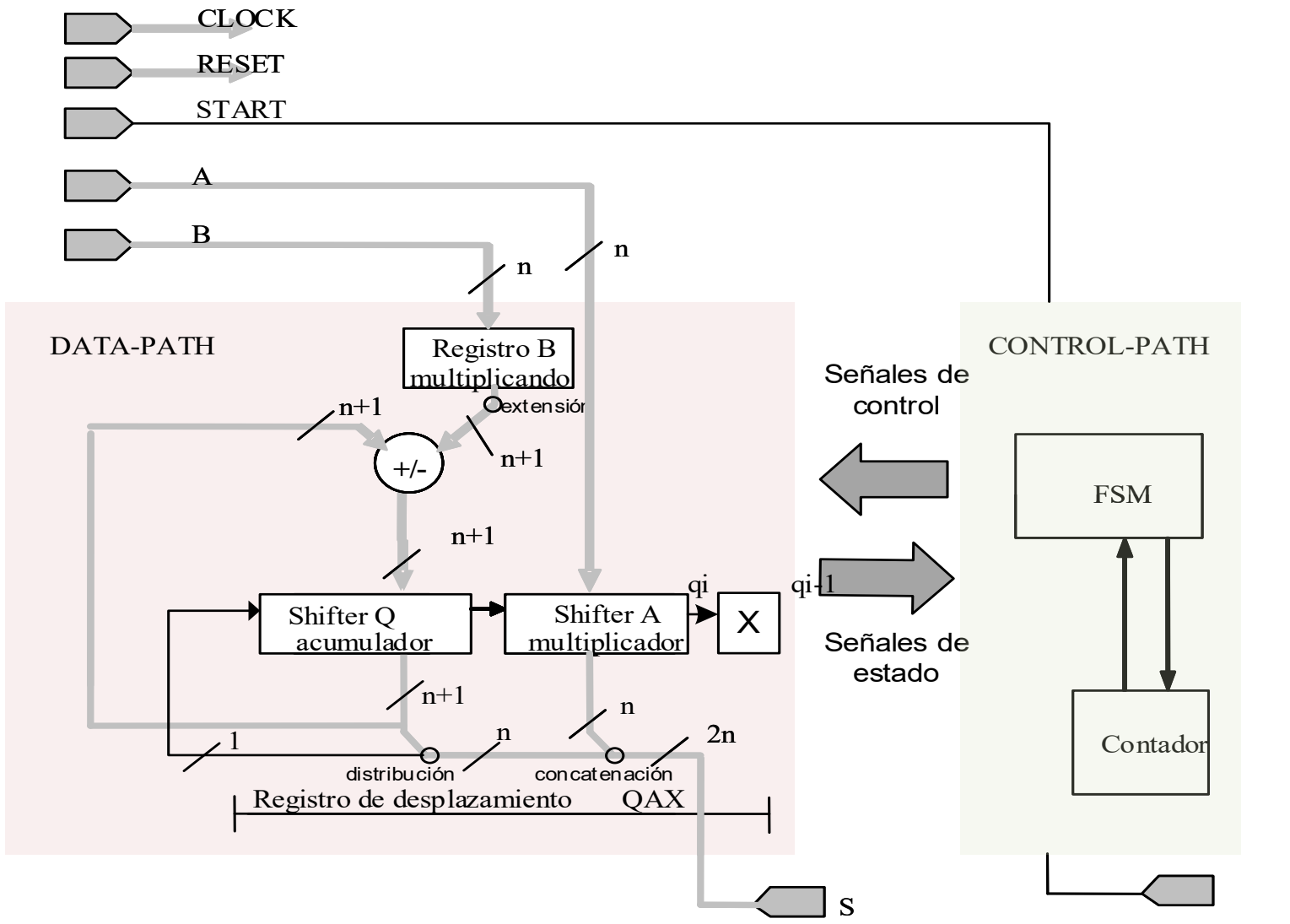


Diagrama de bloques de la solución



El algoritmo the Booth modificado

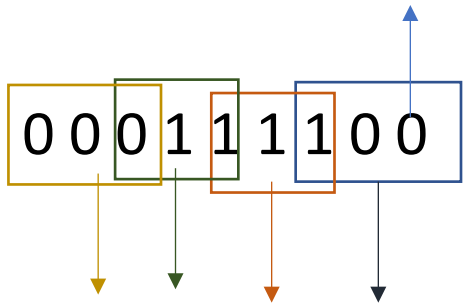
El Algoritmo the Booth modificado

- Recodificación por **parejas de bits** para reducir a la mitad el numero de ciclos

q_{i+1}	q_i	q_{i-1}	Digito de Booth
0	0	0	0
0	0	1	+1
0	1	0	+1
0	1	1	+2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

bit implícito 0 a la derecha del LSB

$30_{10} = 0011110_2 =$ 0 0 0 1 1 1 1 0 0



Desplazo dos posiciones

0 +2 0 -2

Booth modificado

Ejemplo algoritmo de Booth modificado

N= 6

M= 13_{10}= 001101_2

Q= -6_{10} = 111010_2

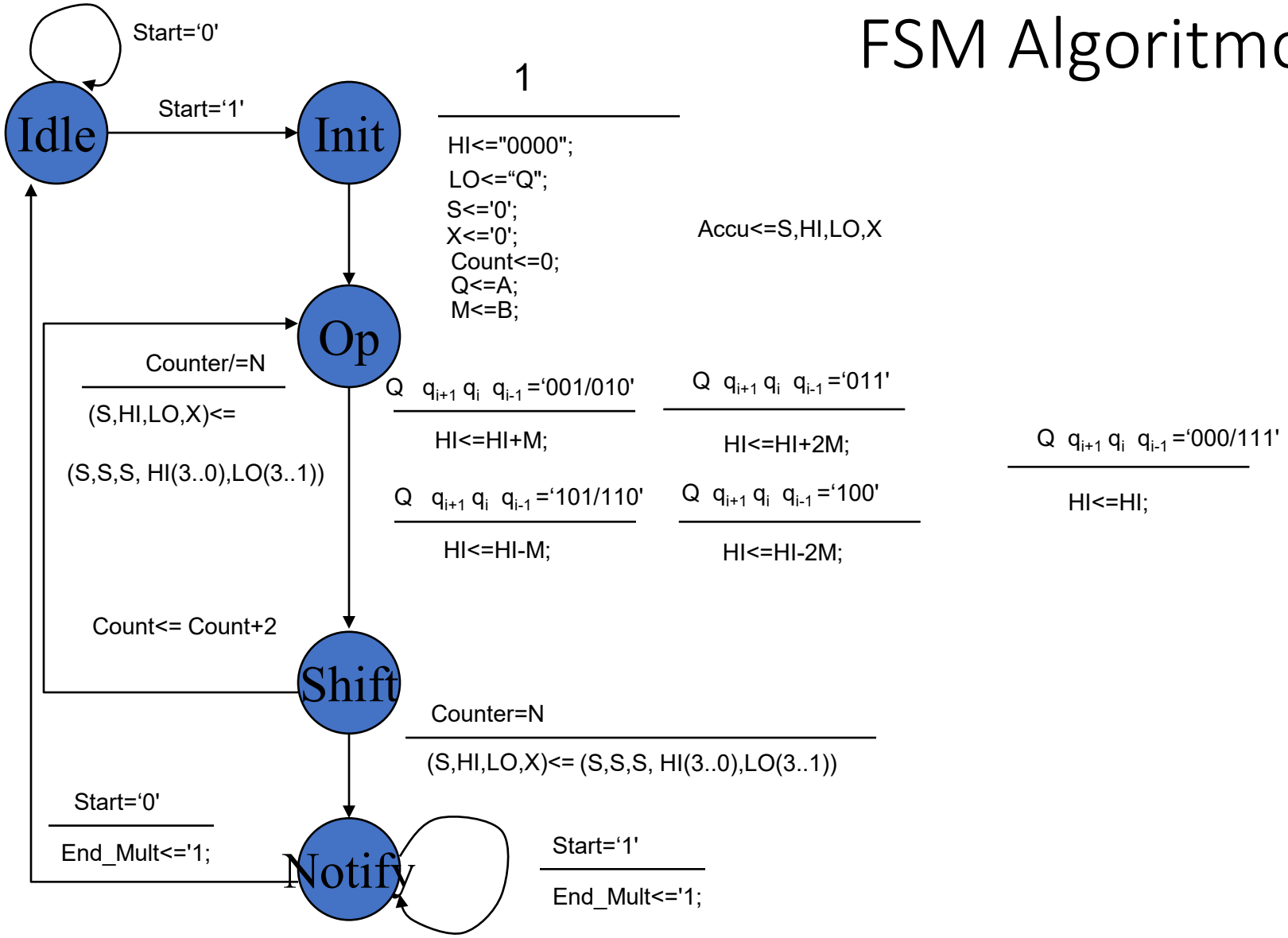
Ciclo	Acción	S - HI - LO - X
0	Carga de valores (LO<=Q ; HI<=0000; S,X<=0	0 000000 111010 0
1	$q_{i+1}\, q_i\, q_{i-1}$ = 100 : HI <- HI –2M con extensión	1 100110 111010 0
	Desplazamiento 2 bits	1 111001 101110 1
2	$q_{i+1}\, q_i\, q_{i-1}$ = 101: HI <- HI - M	1 101100 101110 1
	Desplazamiento	1 111011 001011 1
3	$q_{i+1}\, q_i\, q_{i-1}$ = 111: HI <- HI	1 111011 001011 1
	Desplazamiento	1 111110 110010 1

-78_{10}

```
  0 0 0 0 0 0
- 0 0 1 1 0 1
-------------
  1 1 0 0 1 1
- 0 0 1 1 0 1
-------------
  1 0 0 1 1 0
```

```
  1 1 1 0 0 1
- 0 0 1 1 0 1
-------------
  1 0 1 1 0 0
```

FSM Algoritmo the Booth modificado



ASM Algoritmo the Booth modificado

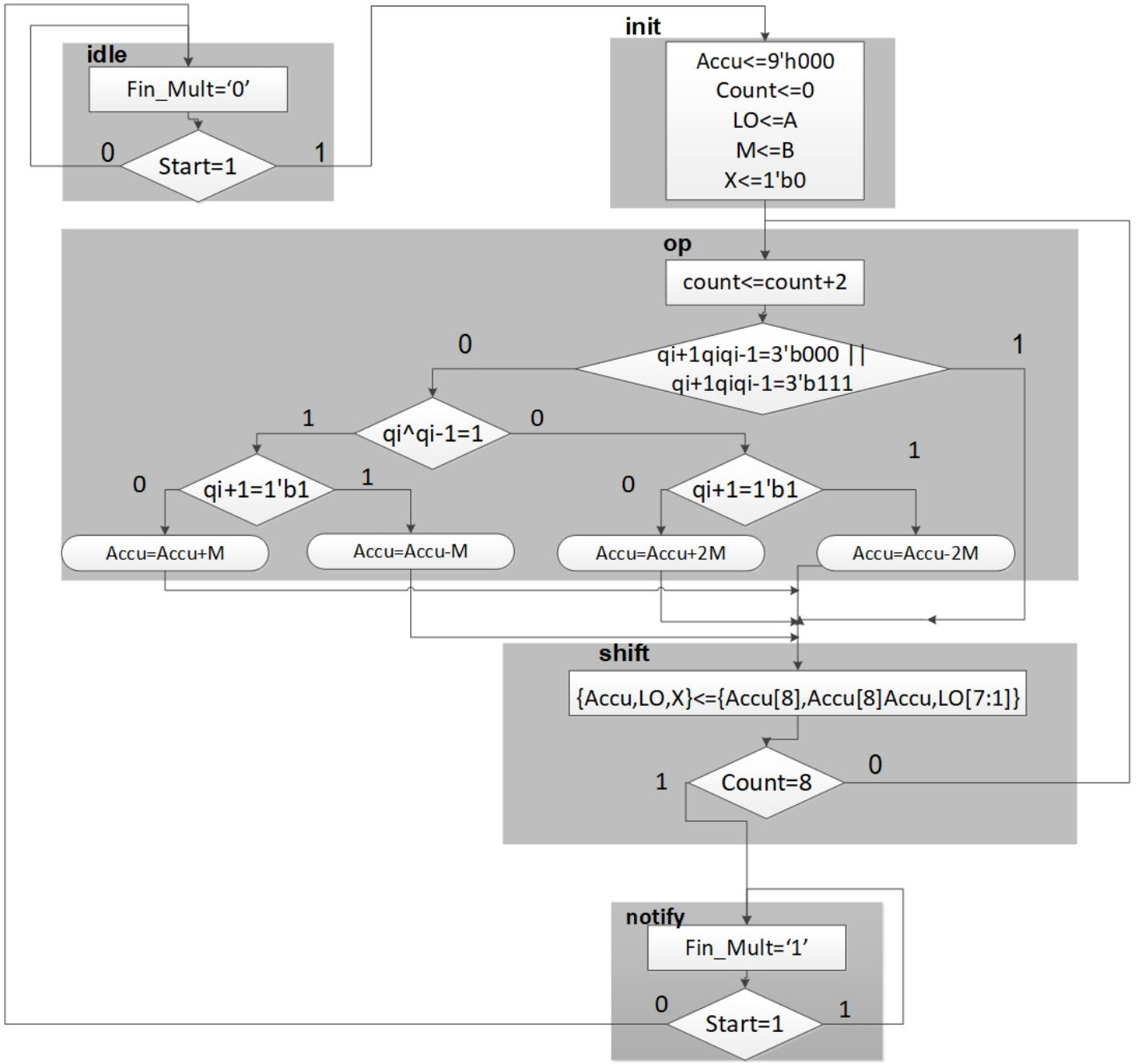
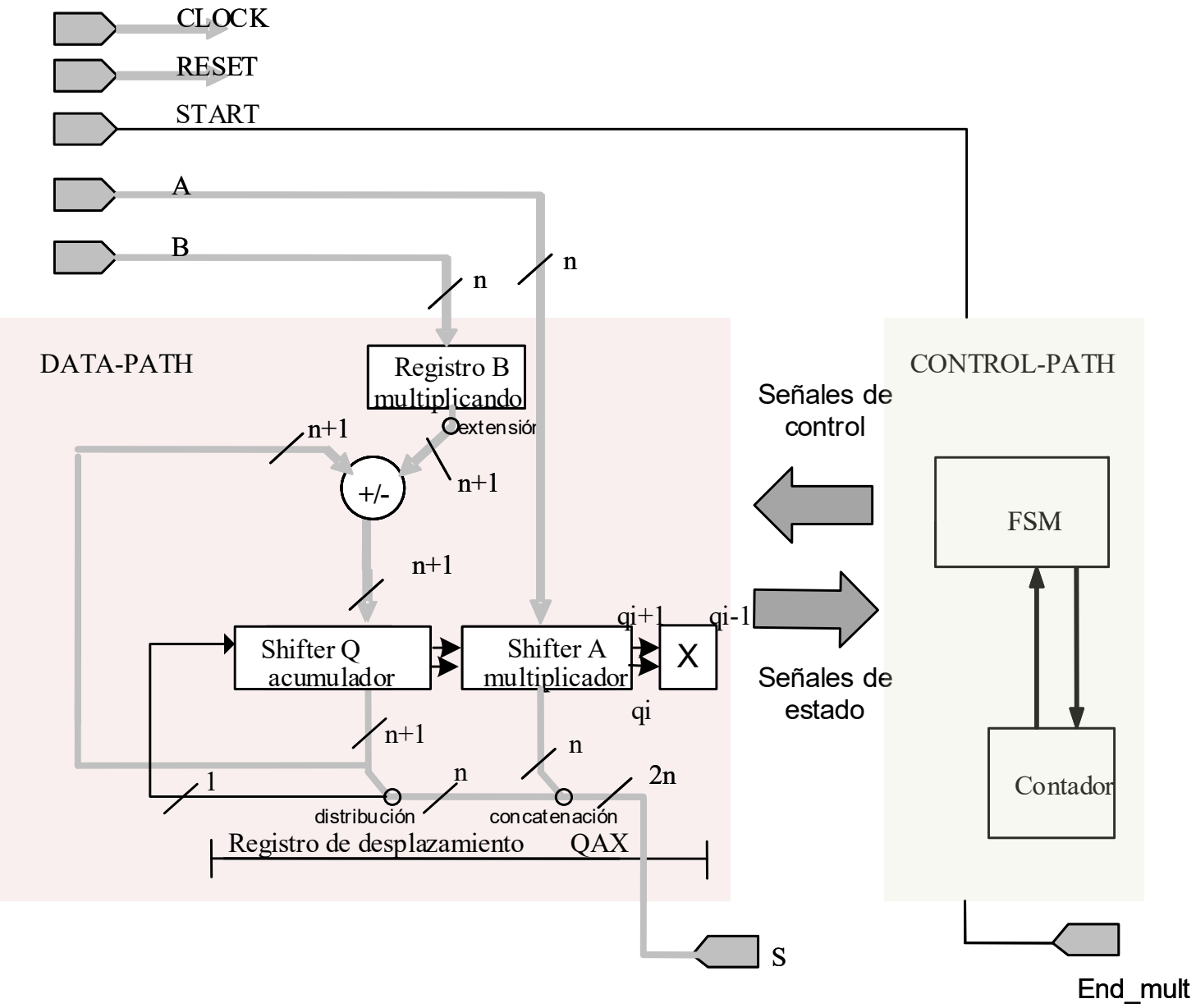


Diagrama de bloques de la solución



SOLO PARA ENTENDER
NO SE PIDE DISEÑO JERARQUICO

Implementación

Un fichero

Systemverilog

Estados como variable enumerada

Un proceso, literal ASM

Asignaciones non-blocking

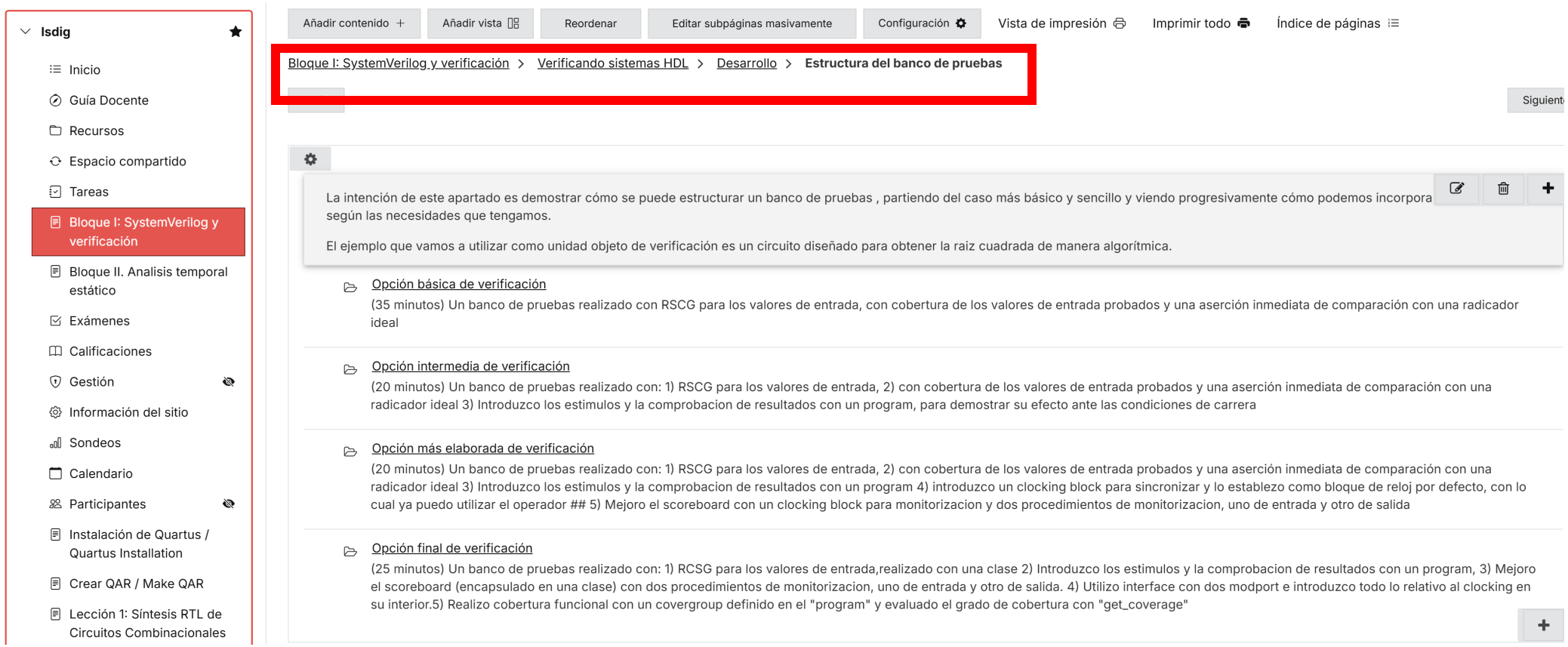
Definir q={LO[1:0],X}

PARTE VERIFICACíON

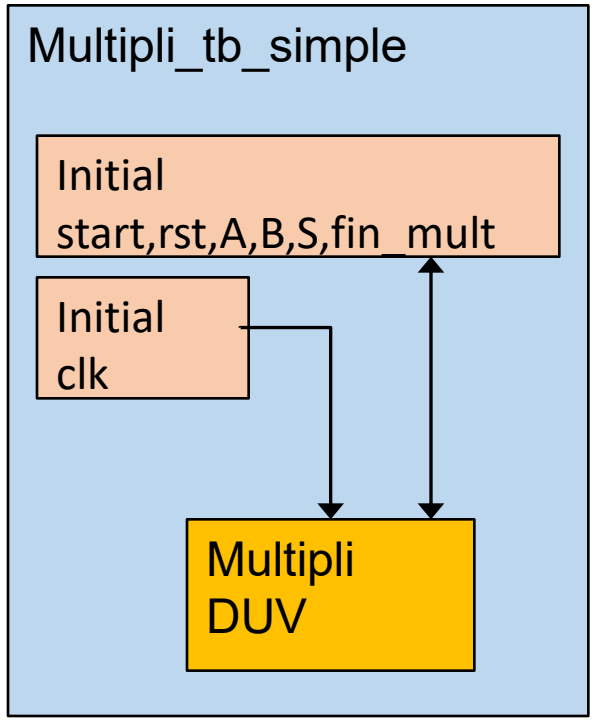
el testbench exhaustivo

El testbench exhaustivo

- Guia completa, con ejemplos, en poliformaT



Verificación Básica – Nivel 0

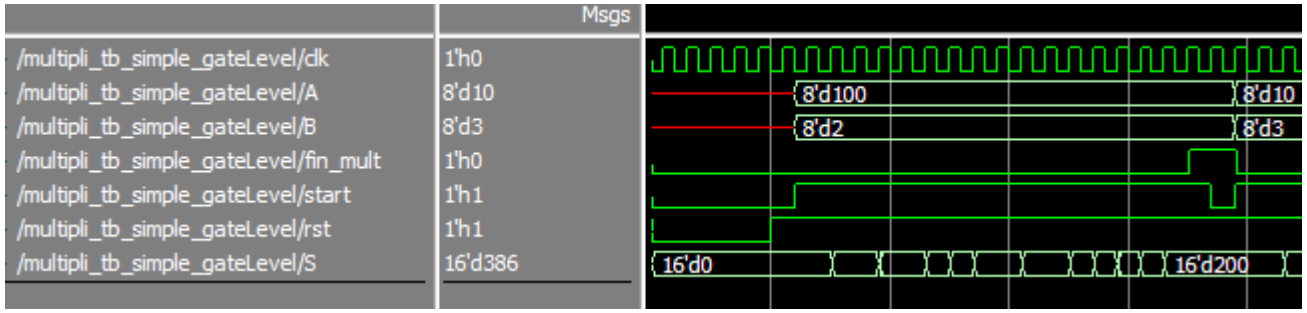


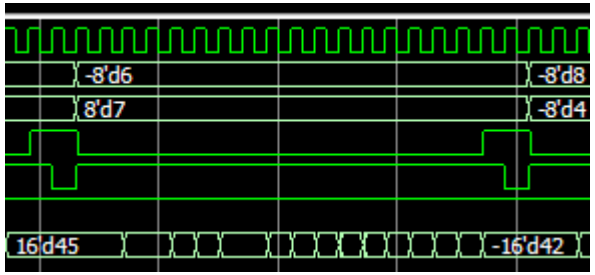

```
`timescale 1 ns / 1 ps
module multipli_tb_simple_gateLevel;

parameter tamano=8;

reg clk;
reg [tamano-1:0] A,B;
logic fin_mult;
logic start;
reg rst;
wire [(2*tamano)-1:0] S;

initial begin clk=1'b0;
forever #50 clk=!clk;
end

initial begin
rst=1;
start= 0;
#1 rst=0;
#500 rst=1;
#100 A=8'd100; //100*2
B=8'd2;
start=1;
@ (posedge fin_mult);
@ (negedge clk);
#50 start=0;
#100 A=8'd10; //10*3
B=8'd3;
start=1;
@ (posedge fin_mult);
```

… resto de casos

```
$stop;
end

multipli DUV
(.CLOCK(clk),
 .RESET(rst),
 .START(start),
 .A(A),
 .B(B),
 .END_MULT(fin_mult),
 .S(S)
);

endmodule
```

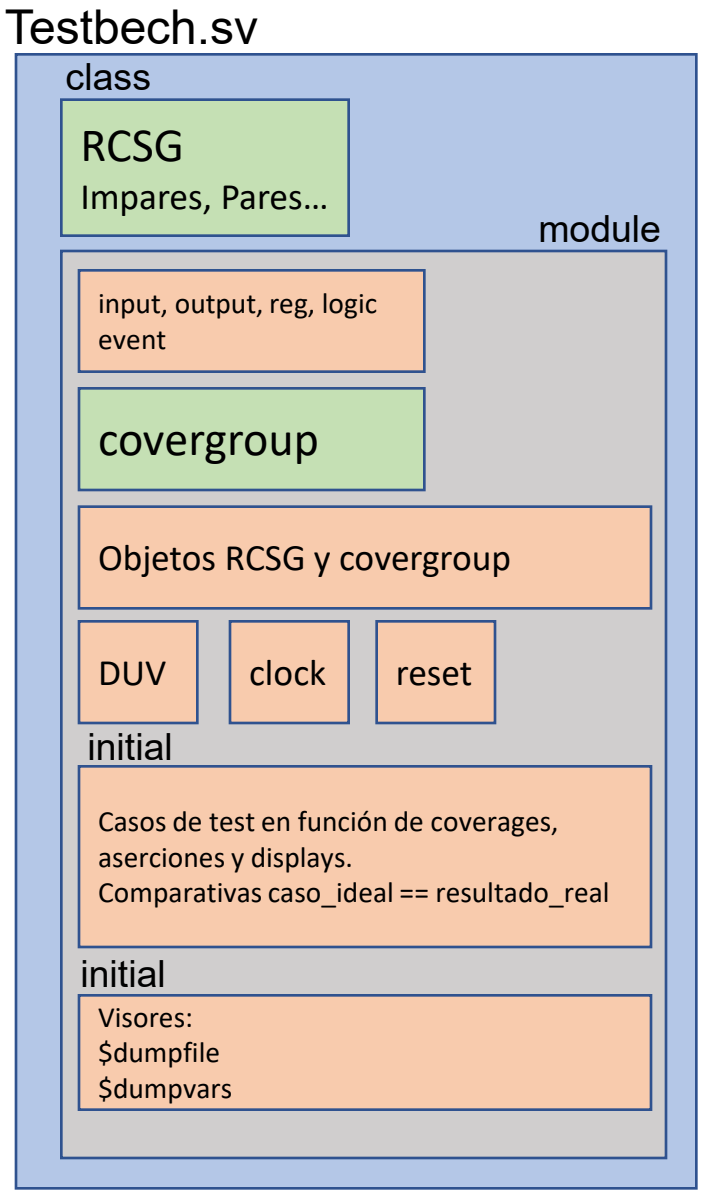
Verificación básica de resultados





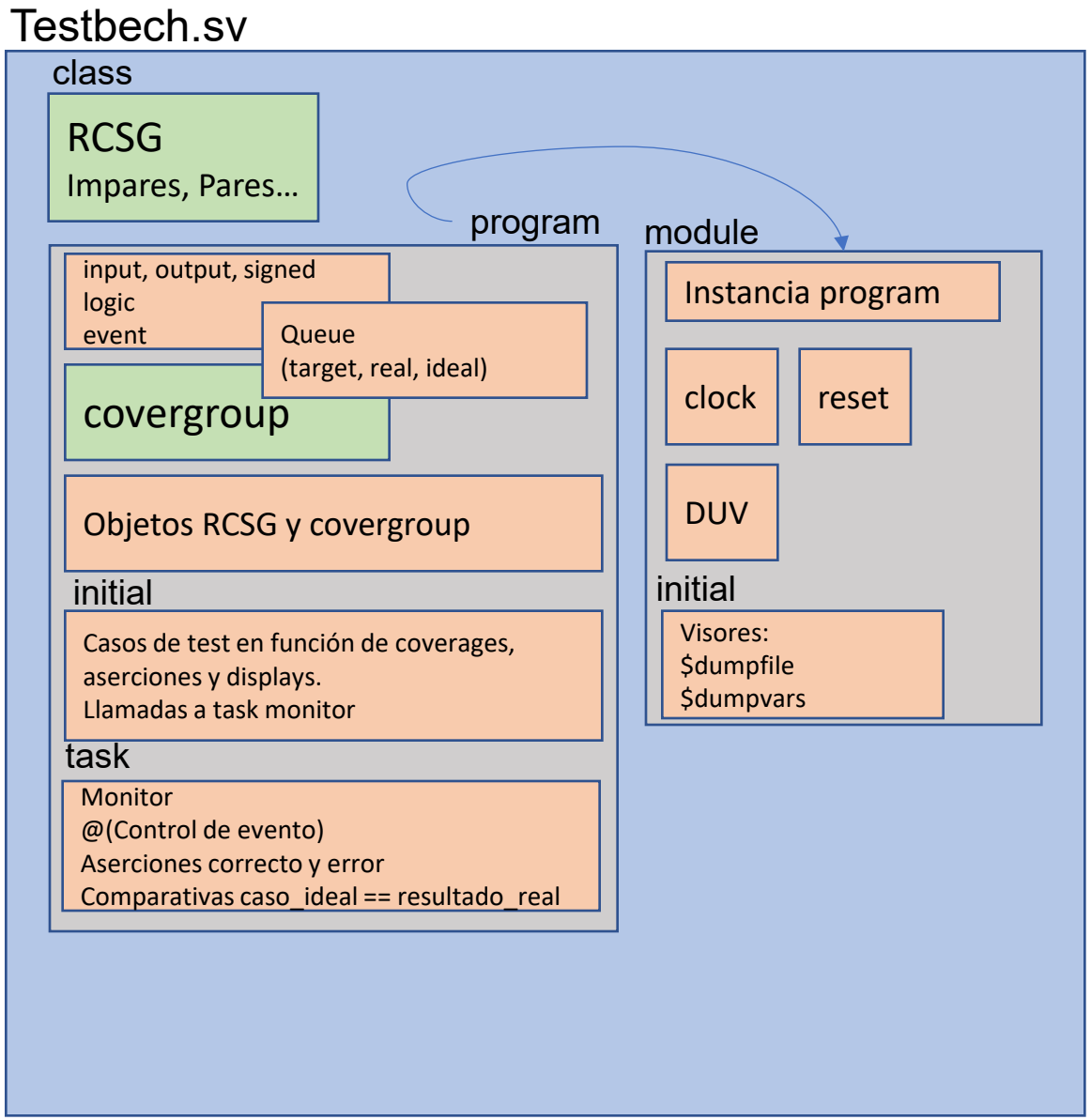
Verificación sin aserciones
Basado en visualización de formas de onda
Muchos casos sin cubrir
No automatizada
No optimizada
Peligro de mal funcionamiento

Verificación Intermedia – Basada en Radicador



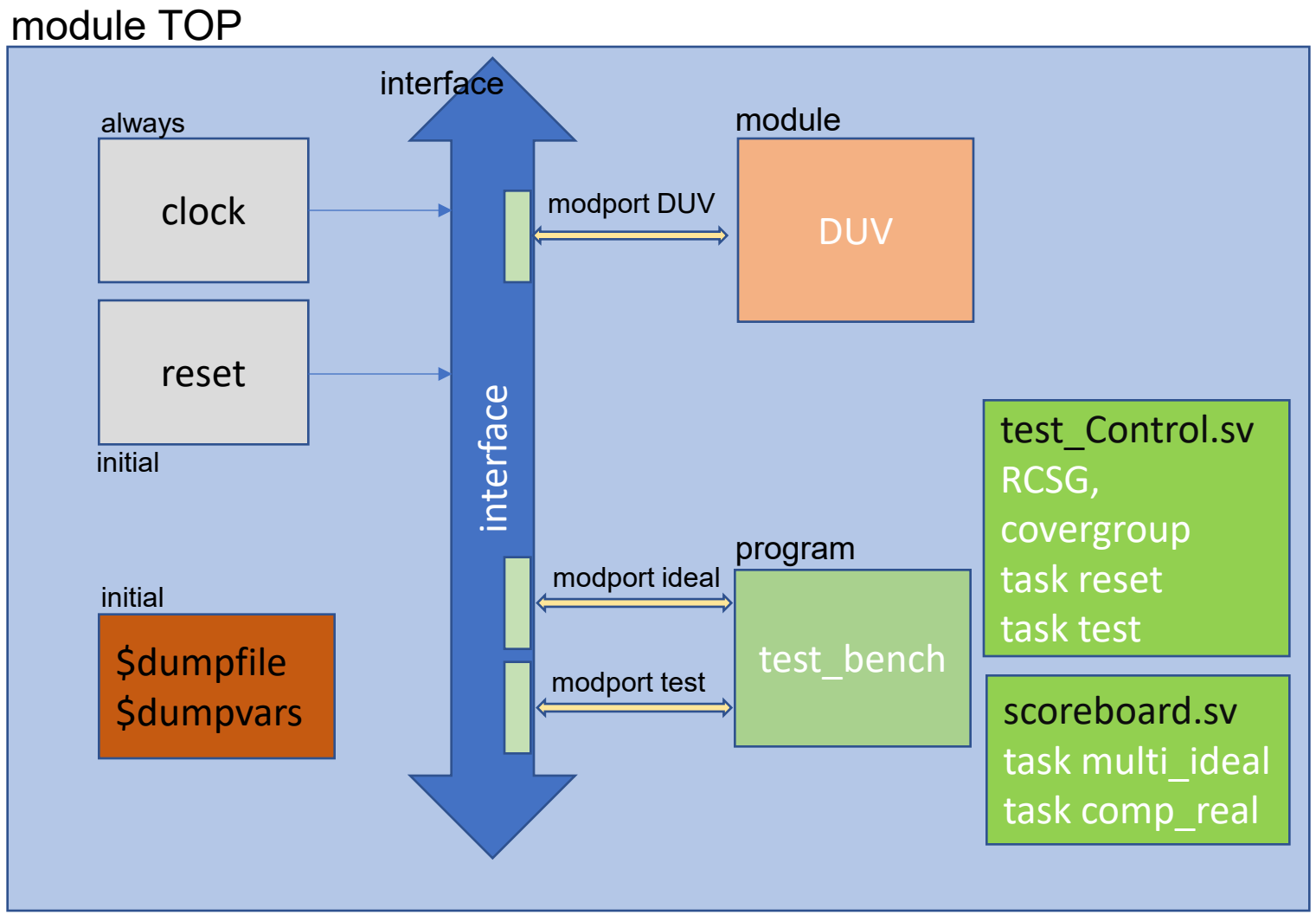
- Características
  - Fichero de test único
  - Incluye generación de estímulos aleatoria RCSG
  - Incluye covergroups
  - Instancia DUV
  - Genera clock y reset en el mismo fichero
  - Los casos de test los agrupa en un initial con aserciones y gestión de eventos para comprobación
  - La visualización de datos con initial al final del fichero
- Fichero largo y poco genérico
- Muestra las posibilidades futuras de estructurar la verificación

Verificación Avanzada – Basada en Radicador



- Características adicionales
  - Agrupación de testbench en PROGRAM
  - Instancia program en module
  - Introduce colas para tratar valores ideales A*B.
  - Uso básico de colas sin embargo
- Todavía un único fichero .sv
- Escaso aprovechamiento de clases.
- No se utilizan interfaces para facilitar la conectividad

Verificación Final - Basada en Radicador



EMPEZAMOS

Sugerencia de organización

1.-Leed bien la tarea y comprended las sub-tareas que hay que realizar

2.-Distribuid en sub-tareas en las 4 sesiones que disponemos para la Tarea II.

3.-Subtareas:

- **3.1**- Desarollo hardware
  - descripción en 1 o 2 procesos
  - verificacion basica
  - segmentación (SEMANA 4)
  - placa (SEMANA 4)
- **3.2**- Desarrollo testbench verificación
  - basica
  - intermedia (SEMANA 2)
  - avanzada (SEMANA 2-3)
  - final (SEMANA 3)

PARTE VERIFICACIÓN

implementación paso a paso
del testbench exhaustivo

Verificación intermedia

- En la verificación intermedia se pide implementar un Testbench estructurado:
  - Utilización de estructura program
  - Utilización de tasks

- Para:

Generación de estímulos aleatorios

Nivel de cobertura funcional

Comprobación del funcionamiento

Generación de estímulos aleatorios

- Objetivo: generar situaciones aleatorias controladas
- Alcance: Multiplicando y multiplicador

(ejemplo divisor con pares e impares (ejem…)

```
class Numeros;
  randc logic [7:0] valorA;
  randc logic [7:0] valorB;
  constraint pares    {valorA[0] == 1'b0 && valorB[0]==1'b0;}
  constraint impares  {valorA[0] == 1'b1 & valorB[0]==1'b1;}
  constraint Apar_Bimpar    {valorA[0] == 1'b0 & valorB[0]==1'b1;}
  constraint Aimpar_Bpar    {valorA[0] == 1'b1 & valorB[0]==1'b0;}
endclass
```

```
Numeros numeros_rcsg;

numeros_rcsg = new;
```

```
// fijo las contraints
$display ("Empiezo la verificació PARES" );
numeros_rcsg.pares.constraint_mode(1);
numeros_rcsg.impares.constraint_mode(0);
numeros_rcsg.Apar_Bimpar.constraint_mode(0);
numeros_rcsg.Aimpar_Bpar.constraint_mode(0);

//aleatorizo y asigno valores
assert (numeros_rcsg.randomize()) else    $fatal("randomization failed");
A= numeros_rcsg.valorA;
B= numeros_rcsg.valorB;
```

Nivel de cobertura funcional

- Objetivo: saber cuantas posibilidades he probado
- Alcance: 100%

```
//Cobertura
covergroup Valores;
  cp1: coverpoint  A {bins binsA[256]={[0:255]} ;}
  cp2: coverpoint  B {bins binsB[256]={[0:255]} ;}
  cp3: cross cp1,cp2;
endgroup;
```

```
Valores valores_cg;

valores_cg=new;
```

```
while ( valores_cg.cp3.get_coverage()<25)

valores_cg.sample();
```

A	B
P	P
I	I
I	P
P	I

Generación de estímulos aleatorios

Nivel de cobertura funcional

Comprobación del funcionamiento

```
class Numeros;
randc logic[7:0] valorA;
randc logic[7:0] valorB;

constraint pares {valorA[0]== 1'b0 && valorB[0]==1'b0;}
constraint impares {valorA[0]== 1'b1 && valorB[0]==1'b1;}
constraint Apar_Bimpar {valorA[0]== 1'b0 && valorB[0]==1'b1;}
constraint Aimpar_Bpar {valorA[0]== 1'b1 && valorB[0]==1'b0;}
endclass
```

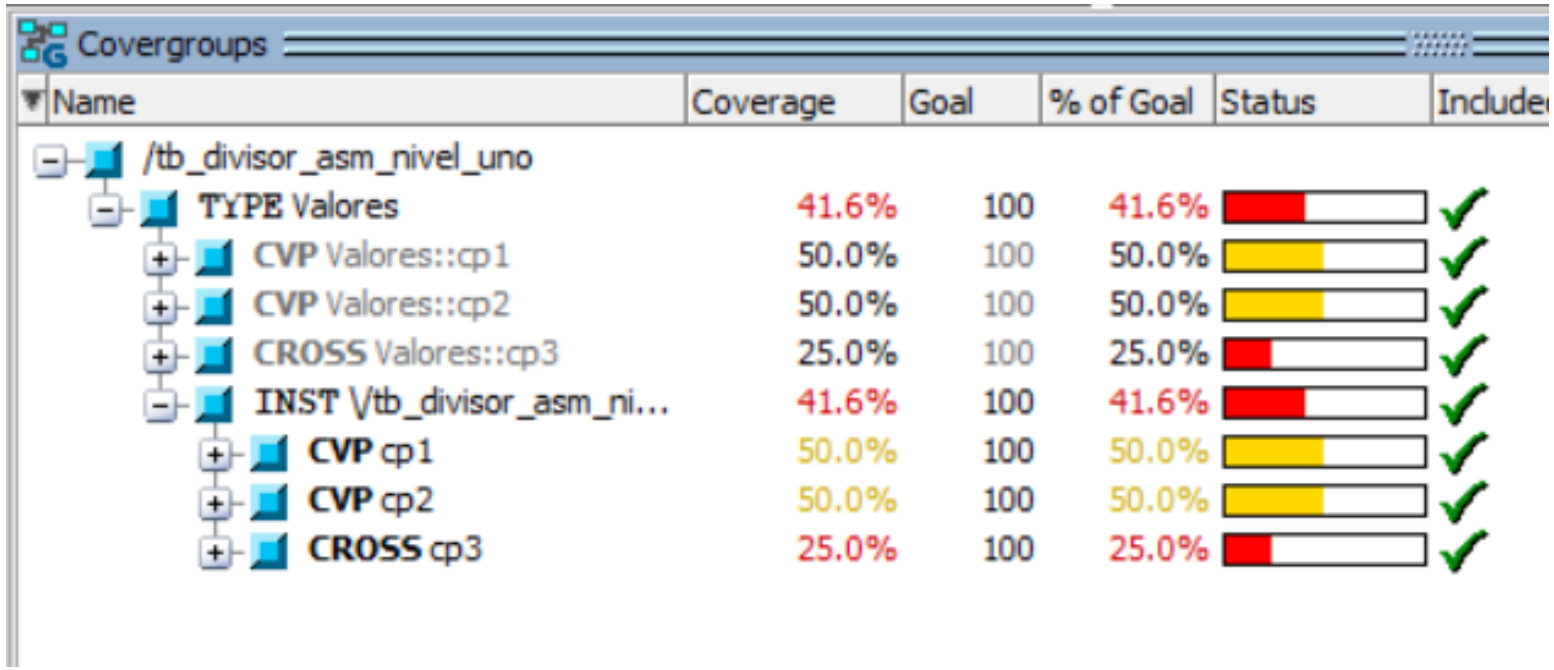
```
Numeros numeros;
Valores valores;
```

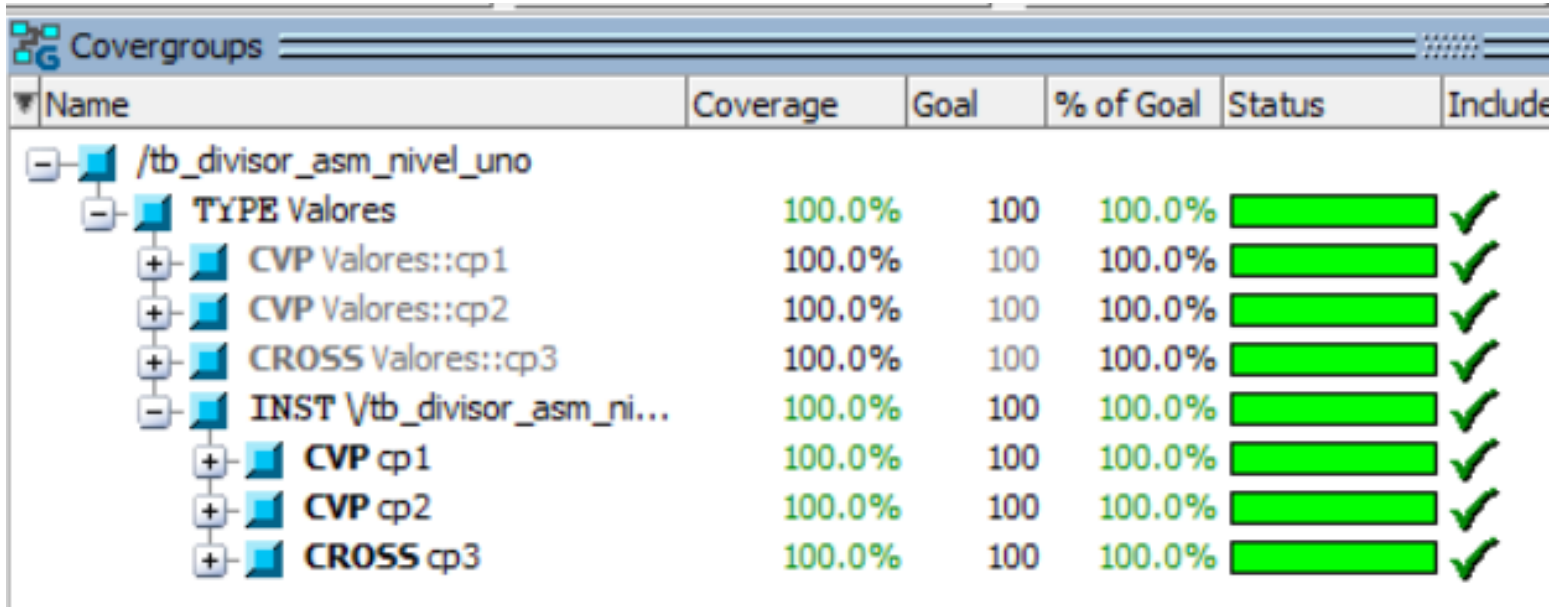
```
numeros=new;
valores=new;

//PARES
$display ("Empiezo con los pares");
numeros.pares.constraint_mode(1);
numeros.impares.constraint_mode(0);
numeros.Apar_Bimpar.constraint_mode(0);
numeros.Aimpar_Bpar.constraint_mode(0);
```

```
while(valores.cp3.get_coverage()<25)
begin
        assert (numeros.randomize()) else $display ("Error al randomizar");
        div (numeros.valorA,numeros.valorB);
        valores.sample();
        $display(valores.cp3.get_coverage());
end
```

Generación de estímulos aleatorios

Nivel de cobertura funcional

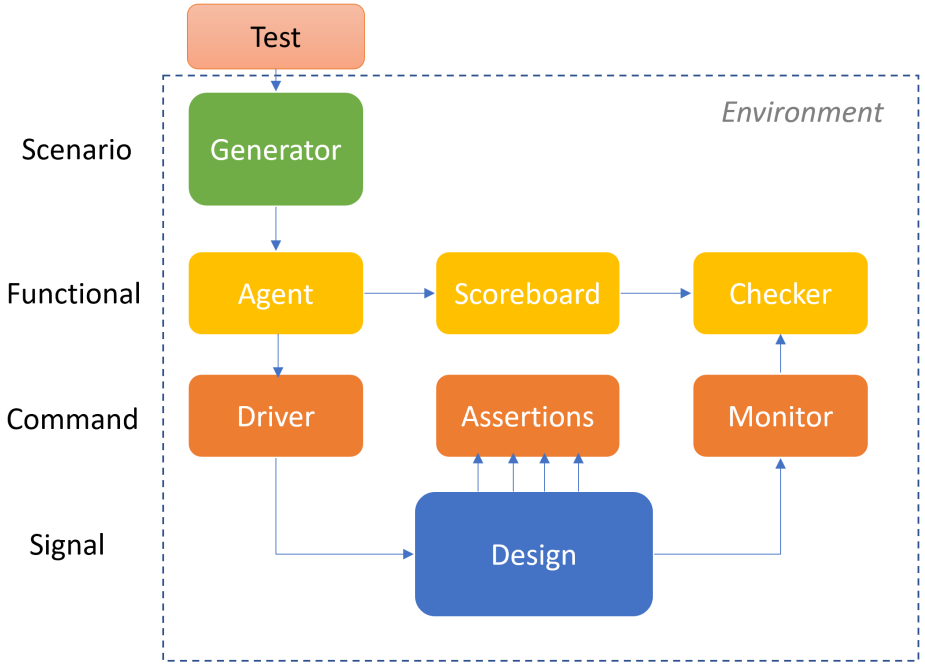




Verificación intermedia

- En la verificación avanzada se pide dar estructura a nuestro testbench mediante la separación de TEST y COMPROBACIÓN

- Para:



Generación de estímulos aleatorios

Nivel de cobertura funcional

Comprobación del funcionamiento

Estructura

Comprobación estructurada

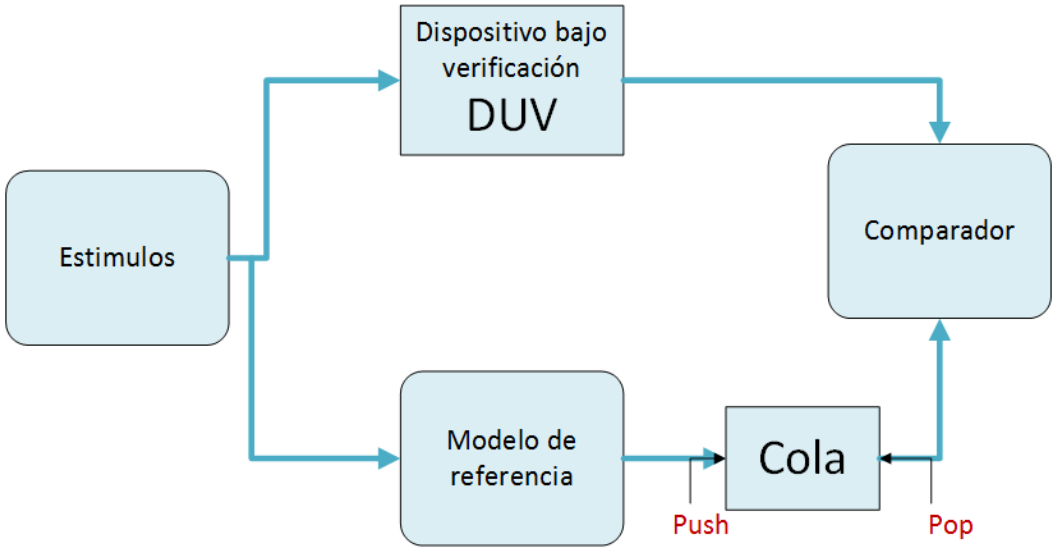
- Objetivo: saber si el diseño funciona como debería
- Alcance: comprobar que en el 100% de los casos se divide bien

Comprobación con aserción

Ubicamos una aserción y comprobamos si han pasado en todos los casos
(P-P, N-N, P-N, N-P)

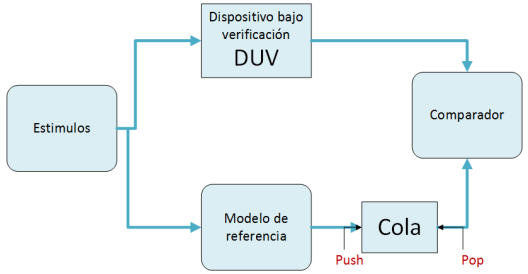
```
Res=A*B;

Assert (S== Res) else $display("Error en multiplicación);
```

Comprobación mediante modelo de referencia



multipli_parallel.sv

Comprobación estructurada




```
logic [tamano-1:0] COC_GM, RES_GM;
```

```
divisor_asm  #(.tamano(tamano)) duv (CLK,RST_n,START,NUM,DEN,DONE,COC,RES);
Divisor_Algoritmico_pruebas #(.tamanyo(tamano)) Golden_Model (CLK,RST_n,START,NUM,DEN,COC_GM,RES_GM,);
```

```
event comprobar;

logic [tamano-1:0] cola [$];

div(numeros.valorA,numeros.valorB);

//COC_GM = numeros.valorA/numeros.valorB;
//RES_GM = numeros.valorA%numeros.valorB;

//comprobación
@(posedge DONE);
cola.push_front(COC_GM);;
cola.push_front(RES_GM);
@(posedge CLK);

->comprobar;
```

monitor;

```
task monitor;
  fork
    while (1)
      begin

        @(comprobar);
        $display("Compruebo valor");
        targetCOC= cola.pop_back();
        targetRES=cola.pop_back();

        if(DEN!=8'd0) assert (COC==targetCOC && RES==targetRES) else $display("operacion mal realizada:

        end
  join_none
endtask
```